USING ADA TO MAXIMIZE VERBATIM SOFTWARE REUSE

Michael E. Stark, NASA/Goddard Space Flight Center
Eric W. Booth, Computer Sciences Corporation

1. INTRODUCTION

The reuse of software holds the promise of increased productivity and reliability. Experience has shown that making even the slightest change to a "reused" piece of software can result in costly, unpredictable errors [Solomon, 1987]. For this reason, the Flight Dynamics Division (FDD) of Goddard Space Flight Center (GSFC) is concentrating effort on developing "verbatim" reusable software components with Ada, where *verbatim* means that no changes whatever are made to the component.

This paper presents the lessons learned on several simulator projects in the FDD environment that exploit features of the Ada language, such as packages and generics, to achieve verbatim reuse. These simulators are divided into two separate, but related, problem domains. A *dynamics simulator* is used by the FDD mathematical analysts to verify the attitude control laws that a spacecraft builder has developed. A *telemetry simulator* generates test data sets for other mission support software. FDD began using Ada in 1985 with the development of the Gamma Ray Observatory attitude dynamics simulator (GRODY). Since that time, six additional simulator projects have been started. With each successive project, a concentrated effort is made to use the lessons learned from previous Ada simulator development projects.

This paper focuses on the concepts used in the projects that have had the most impact on verbatim software reuse in the FDD environment: GRODY, the Upper Atmosphere Research Satellite Telemetry Simulator (UARSTELS), and the Generic Dynamics and Telemetry Simulator (GENSIM). This paper defines underlying design principles, discusses how Ada features support these principles for reuse in the small, and shows how these principles are used to achieve reuse in the large. Finally, this paper presents supporting data from current reusability results.

The FDD has been using a modified version of the General Object-Oriented Development (GOOD) methodology [Seidewitz, 1986; Stark, 1987; Seidewitz, 1988] to develop its Ada software. Three concepts that play a role in GOOD enhance verbatim reuse: abstraction, inheritance, and problem-specific architectures. These concepts support the reuse of successively larger components within successively narrower domains. The next two sections describe how these concepts are applied to simulator projects in the FDD. Abstraction supports "reuse in the small," and inheritance and problem-specific architectures support "reuse in the large." The current practice is to support reuse in the small through component libraries, as is done with a collection of components by Grady Booch [Booch, 1987] and EVB's Generic, Reusable Ada Components for Engineering (GRACE) [Berard, 1989]. The UARSTELS and GENSIM projects are cited to describe how reuse on a large scale is accomplished and to demonstrate the potential in cost savings and/or the ability to solve more complex problems.

2. REUSE IN THE SMALL: USE OF ADA GENERICS

This section presents design and implementation guidelines for using Ada generics. It shows how the design principles mentioned in the previous section are implemented using Ada.

Designing individual generic components is understood to the point that such components are commercially available. The Booch taxonomy of structures creates a family of components that satisfy the same abstraction within the context of different problems, for example, sequential versus concurrent applications.

The design process becomes more interesting when a hierarchy of generic components is needed. This is the case when designing generic subsystems with multiple levels of abstraction. This leveling of a subsystem can be embodied in the Ada code by using generic units and instantiations in the following three ways:

1. Library unit instantiations
2. Nested instantiations
3. Nested generic definitions

This section will define each of these approaches and provide examples to demonstrate their application. Implications for reusability and lessons learned are included for each approach.

Library Unit Instantiations

The first approach is to create an instantiation of a generic that is a library unit. This approach is appealing for practical reasons. The potentially broad scope provided by library unit instantiations may be necessary for Ada compilers that do not implement code

sharing[1] [Ganapathi, 1989]. Most implementations currently do not support code sharing. Instead, each instantiation creates a complete object code copy of the generic template. The system-wide scope provided by library units often makes only one copy (instance) necessary.

To describe the first method of library unit instantiations, assume that the generic package A depends on a set of subprograms, P, provided by the generic package B. Instantiating B as a library unit creates a copy of this generic package called Instance_B. The generic package A may then be instantiated by using the subprograms provided by Instance_B as actual parameters. This allows the generic packages A and B to be designed and implemented having no external dependencies, which makes reuse simpler. This approach is depicted in the design diagram[2] shown in Figure 1.

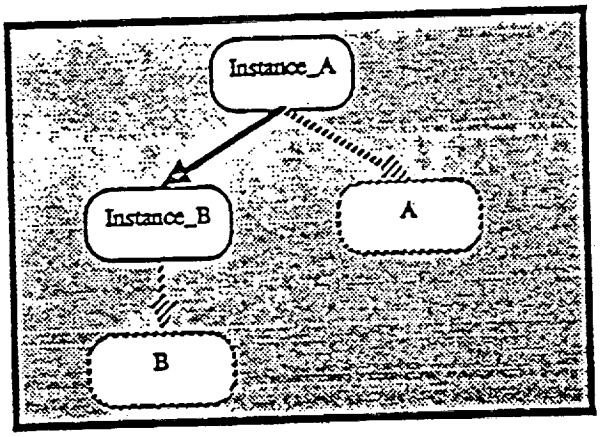

Figure 1.

As previously mentioned, instantiating generics as library units has the advantage that instantiations of A and B may be imported (named in a with clause) by any compilation unit in a system.

When full visibility is desired, this technique works well. However, if B were an abstract state machine[3] and the design required that B should be visible only to A, this potentially broad visibility of B is undesirable. It would be far better to use the language to enforce that design decision.

Another drawback of library unit instantiations is that as generic components become more complex, they require a longer list of more problem-specific generic formal parameters. Each instantiation of the generic becomes long and complex. The instantiations can be made simpler by specifying defaults for formal subprograms using the notation "with procedure P1 is <>." Then (assuming in this case that A's only formal parameter is P1), the instance of A can be written as shown in Figure 2.

```
generic
  with procedure P1 is <>;
package A is
  ...
end A;

generic
  ...
package B is
  procedure P1;
  ...
end B;

with A,Instance_B;
use Instance_B;
package Instance_A is new A;
```

Figure 2.

This technique works well when numerous simple functions are used as formal parameters. This use of Ada *simulates* the search of an object code library to resolve external references. Figure 3, part 2 presents an example of mathematical packages being used to instantiate the package of flight dynamics abstract data types[4] shown in part 1. Care should be used with this technique, since the use clause does more than simply make objects and operations directly visible. There are visibility and precedence rules of the language that will affect what defaults will be used [Mendal, 1988].

```
generic
   type REAL is digits <>;
   type RADIANS is digits <>;
   type VECTOR is array ( INTEGER range <>) of REAL;
   type MATRIX is array ( INTEGER range <>,
                          INTEGER range <>) of REAL;
   with function sin ( Angle : in RADIANS)
        return REAL is <>;
   with function cos ( Angle : in RADIANS)
        return REAL is <>;
   with function Floor ( Item: in REAL)
        return REAL is <>;
package Generic_Attitude_Types is
   ...
end Generic_Attitude_Types;
```

Figure 3 (1 of 2).

[1] Code sharing is a technique that allows each instance of a generic to share the same object code. The result is usually a smaller object code size and slower execution speed for the system.

[2] The notation for the design diagrams uses rounded-corner rectangles to represent packages, solid arrows to represent dependencies, broken arrows to represent instantiations, and broken package symbols to represent a generic unit.

[3] A package that is an abstract state machine is a package that maintains state information in the package body [Booch, 1983].

[4] A package that is an abstract data type exports objects, types, and operations but does not maintain state information in the body [Booch, 1983].

```
with  Single_Math_Functions,
      Single_Linear_Algebra;
use   Single_Math_Functions,
      Single_Linear_Algebra;
with  Math_Types,
      Generic_Attitude_Types;

package Single_Attitude_Types is
  new Generic_Attitude_Types (
    REAL     => Math_Types.SINGLE,
    RADIANS  => Single_Math_Functions.RADIANS,
    VECTOR   => Single_Linear_Algebra.VECTOR,
    MATRIX   => Single_Linear_Algebra.MATRIX );
```

Figure 3 (2 of 2).

A second method that uses library unit instances is to instantiate B as a library unit, which is then with-ed into the body of generic A (Figure 4). Since the procedures do not need to be passed as actual parameters, this option allows A to have a shorter formal parameter list. However, method often requires the use of common types for A and B. For example, for a flight dynamics application, the generic package A would have to be coupled to the same floating-point types as the generic or instance of package B. As long as this sort of coupling is relatively simple, it can be managed (in the simulator case by having a single package containing the basic floating-point types).

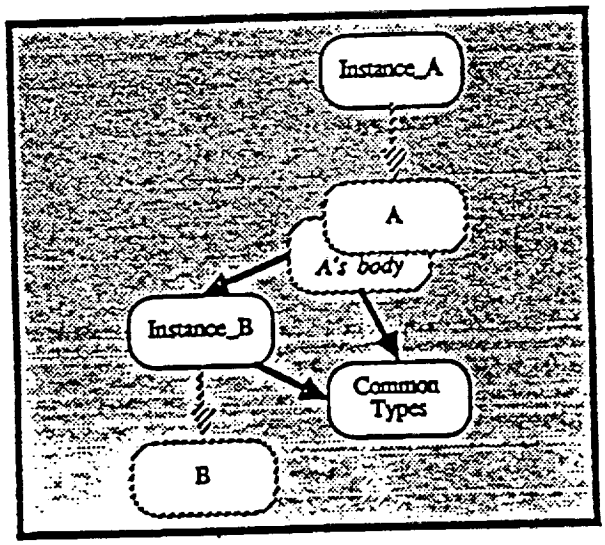

Figure 4.

The disadvantage of this approach is the use of a common types package to implicitly couple A and B. This defeats software engineering principles of information hiding and data abstraction. The following section describes how to exploit these software engineering principles by nesting generic instantiations.

Nested Instantiations

Continuing the same example, the generic package B may be instantiated in the body of generic A (Figure 5) using the generic formals of A. This option is ideal for abstraction and information hiding because it can be extended to a series of nested instantiations. Objects, types, and operations from B may be used as building blocks and specialized to raise the level of abstraction [Stark, 1987]. This is sometimes referred to as re-exported. Importing the package B in this manner allows objects, types, and operations to be hidden in the body of the generic package A but still used to instantiate B.

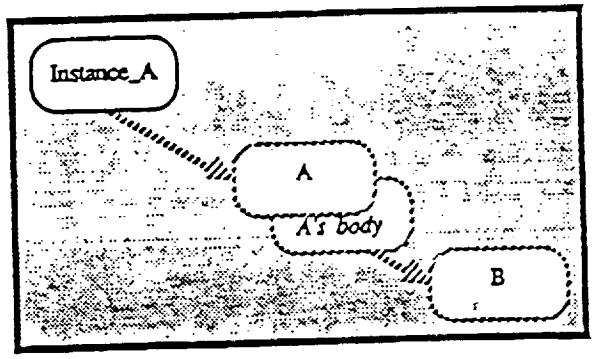

Figure 5.

Using nested instantiations is an appealing technique for software engineering reasons. The limited scope provided by a package boundary increases information hiding and protection. The amount of information that the user of the outer package A needs to know may be limited to the package specification of A. The fact that an instantiation of a lower level generic is used to implement the package is irrelevant to the user. Ramifications of future changes made during the maintenance phase will be much more limited in scope than the library unit instantiation approach.

The disadvantage of using nesting instantiations is the advantage of using library unit instantiations. That is, if the Ada compiler being used does not support code sharing and an instance of a particular generic is necessary in several locations, nesting will result in multiple object code copies. For example, the executable size of GRODY is less than 2 megabytes (Mb). This simulator does not have multiple copies of nested instantiations. UARSTELS, on the other hand, makes extensive use of nested instantiations, which resulted in an executable size of 6 Mb.

On the surface, the implication of these findings seems to suggest using library unit instantiations exclusively. The actual implication, however, is that one should use nested instantiations only when a few copies are necessary. One way to minimize the number of copies is to implement the generic package as an abstract data type rather than an abstract state machine. This approach may be possible when multiple instances of the abstraction use the same types and subprograms as actual parameters but use different objects (Ada allows objects to be passed at run time, while types and subprograms must be passed at instantiation time). This approach allows copies to be created with object declarations instead of generic instantiations; it also has the benefit of increased flexibility, since objects may be declared static at compilation time or created (dynamically) at run time.

When the correct design calls for multiple instances of an abstract state machine the long-term solution is to acquire a compiler that supports code sharing.

Nested Generic Definitions

Nested generic definitions is the third design approach described here. This technique is appealing when the problem calls for a high degree of coupling between generics.

Changing the previous examples, if instantiations of generic package A and generic package B will have common types and subprograms as actual parameters, the following architectures are possible:

1. Make the types and subprograms visible to the generic templates via with clauses.
2. Make each generic package a library unit and duplicate the generic formal parameters in the generic part of each.
3. Nest the generic definitions within the specification of another generic package, C. The generic part of C contains the common formal parameters (Figure 6).

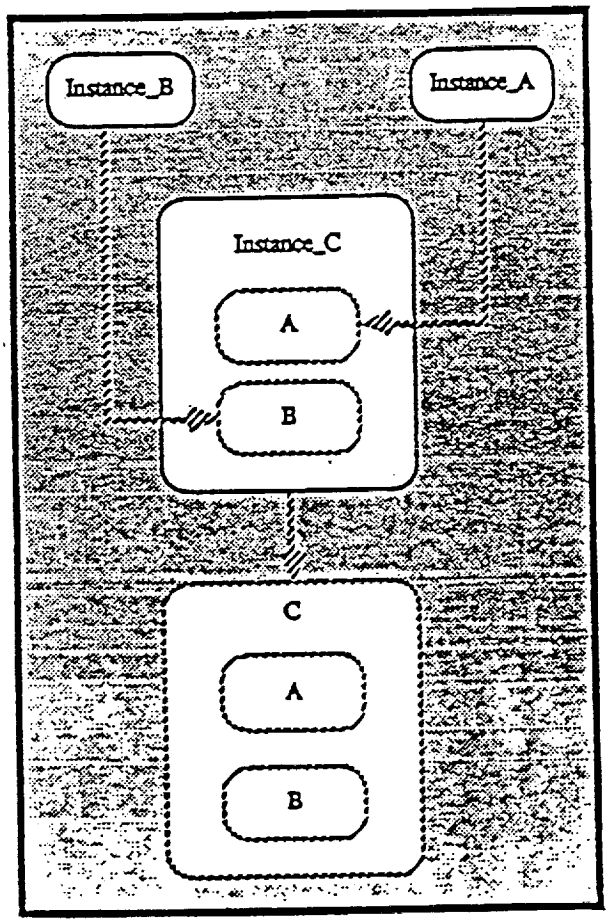

Figure 6.

Although the first option works, it suffers from a high degree of coupling. Future instances of either A or B will always be using the same common types and subprograms. This inflexibility results in limiting the verbatim reusability.

The second option is a large improvement over the first. Future users of packages A and B may now supply their own types and subprograms for the generic formal parameters. However, this architecture becomes tedious and error prone when the common types and subprograms are long and complex. Since generic instantiation becomes a large part of the effort when maximizing verbatim reuse, it is desirable to simplify this activity.

The third option accomplishes the same goals as the second option but with less duplication. This option is useful when the number of nested generics or the number of common generic formal parameters becomes large. It is less error prone because the common actual parameters are supplied only once. The maintenance phase also benefits from the single location of common actual parameters.

Figure 7 shows an example of nested generic definitions from UARSTELS. The generic function FSS Digitize is declared within the generic package Generic Sensor Digitization. The generic formal parameters of the composite package (generic sensor digitization), as well as the generic formal parameters of FSS Digitize, are referenced in the body of FSS Digitize.

```
generic
   type REAL is digits <>;
   type COUNTS is range <>;
   with procedure Log_Error
        (Message : in String) is Text_IO.Put_Line;
package Generic_Sensor_Digitization is

   function Linear_Digitize
     ( Parameter : in REAL;
       Bias : in REAL;
       Scale: in REAL ) return COUNTS;

   generic
     type COUNTER is range <>;
     with function tan (X: REAL) return REAL is <>;
     with function sin (X: REAL) return REAL is <>;
     with function cos (X: REAL) return REAL is <>;
     with function Floor (X: REAL) return REAL is <>;
   function FSS_Digitize
     ( Angle : in REAL;
       Coefficient : in REAL;
       Tolerance : in REAL;
       Maximum_Number_of_Iterations : in COUNTER )
     return COUNTS;
     ...
end Generic_Sensor_Digitization;
```

Figure 7.

Finally, a practical benefit accrues from using the nested generic definition approach. Most compilers, as previously pointed out, do not support code sharing; instead, they expand the generic template at instantiation time. The advantage for the designer needing only 1 generic from a package containing 10 nested generic definitions is that only the object code for the 1 instantiation will be generated.

3. REUSE IN THE LARGE: PROJECT-SCALE REUSE

This section presents the details of the two projects in FDD that have had the most impact on verbatim reuse: UARSTELS and GENSIM. For both projects, an overview is presented giving the background information, the goals, and the motivating factors involved during development. Each system's architecture is discussed using the concepts and notation from the previous section of this paper. Finally, the lessons learned from each project are discussed with their implications for future development efforts.

UARSTELS Overview

The UARSTELS project was started in February 1988 and was the fourth Ada simulator initiated at FDD. Previous simulators included the GRODY experiment, the Geostationary Operational Environmental Satellite-I (GOES-I) dynamics simulator (GOADA), and the GOES-I telemetry simulator (GOESIM). The GOES-I simulators represent the first operational Ada software developed at FDD.

The GRODY design team exploited the feature of nested units, which resulted in increased information hiding (information protection might be a better phrase). The rationale for using information hiding is increased reliability. Higher reliability should, in turn, increase reusability; that is, if a component is very reliable, it is appealing to reuse. This was the basis for the GRODY design.

The lesson learned from this approach was that extensive use of nested packages actually decreased reusability [Quimby, 1988]. In addition, during the coding and testing phases, the development team observed the high compilation overhead incurred by the nested architecture.

Given these lessons from GRODY, the GOADA and GOESIM design teams developed a non-nested architecture with the twin goals of increasing reusability and reducing compilation overhead. Both these goals were met. Individual software components could be picked up by successive projects and reused with slight modifications. The use of library packages, rather than nested packages, kept the compilation costs to a minimum.

One of the lessons learned from the implementation phases of the GOADA and GOESIM projects was that using Ada was not significantly decreasing the level of effort for integration testing. This was unexpected. It was predicted that the integration test phase would require less effort than past FORTRAN integration test phases. Some Ada developments have claimed that system integration took significantly less effort than for similar, previous non-Ada projects [Hudson, 1988].

Analysis by the UARSTELS design team showed that by un-nesting GRODY's packages, more objects and types became visible at a high level in the GOADA design. This increased the number of components to integrate at each level.

UARSTELS Architecture

The architecture of UARSTELS was influenced from the start with the knowledge that another, very similar simulator would follow: the Extreme Ultraviolet Explorer (EUVE) telemetry simulator (EUVETELS). A high level of reuse from UARSTELS to EUVETELS was both desired and thought to be possible because of reused functional specifications. However, because the two spacecraft were themselves different, the telemetry simulators would be different. The design for UARSTELS needed to take into account these spacecraft dependencies and parameterize them.

Each design decision made on UARSTELS attempted to satisfy the following requirements:

1. All UARSTELS requirements
2. Some known requirements from previous systems
3. Some possible future mission requirements

The goals of the UARSTELS team were to maximize *verbatim* reuse and allow the compiler to check system integration as much as possible. To achieve this, the design team took a hybrid approach to the system's architecture. Most packages were developed as library packages, rather than nested units; however, these packages were designed as generic units. This allowed the instantiations of these generic packages to be *nested* in successively higher level packages. The level of nesting (or layering) within UARSTELS is comparable to that of GRODY, with the important difference being the use of generic units in UARSTELS. The generic packages may be picked up and reused just as the nongeneric packages in GOADA or GOESIM, with the important difference again being the use of generics. The nested instantiations allow the language to perform integration checks (whether the programmer wants them performed or not) at each compilation just as in GRODY.

A specific example of this nested instantiation approach is the design of each simulated sensor model within UARSTELS. In Figure 8, the Report_Writer, Data_Set, and Plot_File generic packages are library units. As such, they may be picked up and reused independently of each other. In the case of a sensor model, however, each of these objects is necessary. To provide all three of these abstractions to each sensor model, they are instantiated within the specification of the generic package Sensor_Output. The application-specific parameters are provided to the three low-level generics when they are instantiated. The sensor-specific parameters are generic formal parameters to Sensor_Output. Sensor_Output is then instantiated within the body of the generic sensor model package, with the sensor-specific parameters being provided at that time. The spacecraft-specific parameters are generic formal parameters of the generic sensor model package. The instantiation of the generic sensor model package resolves all the formal parameters, resulting in the Fine_Sun_Sensor object.

As a result of this architecture, UARSTELS differs from previous systems in its recompilation time and executable size. The increased use of generic units had a direct effect on the compilation overhead. It takes longer, in CPU time and elapsed time, to recompile UARSTELS than any of the other simulators. In addition, while UARSTELS is significantly smaller in source lines of code and in number of components, it is also significantly larger in executable image size. This is because the Ada compiler used in FDD does not support code sharing. Instead, it expands generic units for each instantiation.

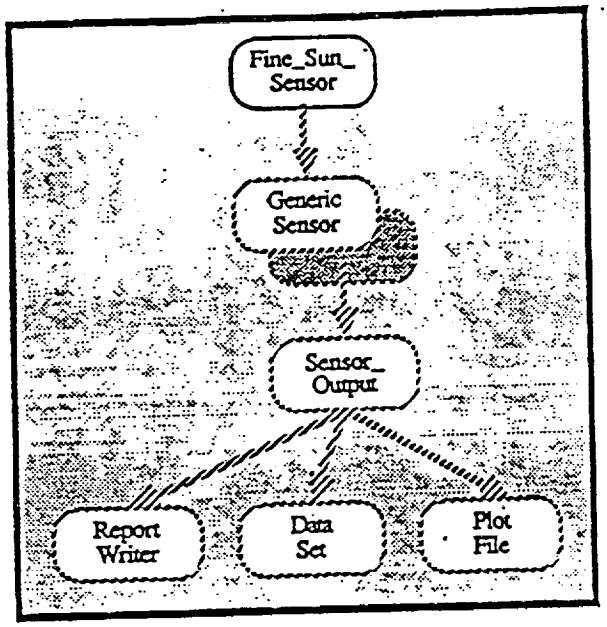

Figure 8.

UARSTELS Lessons Learned

The lessons learned may be summarized as follows:

- Nesting reduces integration testing
- Library units provide reusable software components
- Generic library units provide reusable components and allow information hiding via nested instantiations
- Many Ada implementations incur a large compilation time overhead for the use of nested and generic units
- Many Ada compilers provide a simple implementation of generic units

The nesting feature of Ada must be exploited. The virtue of nesting is information hiding (protection) and higher reliability; this is one of the promises of using Ada. However, like all of Ada's promises, higher reliability does not happen automatically. It must be engineered.

The strong typing feature of Ada must also be exploited. The definition of distinct types that are relevant to the application domain, such as flight dynamics, needs to be engineered. Ada can help eliminate the common mistakes (i.e., mixing radians and degrees or meters and kilometers), but this will not occur automatically either. Through the use of strongly typed objects, reliability can be further improved and integration testing can be further automated.

Strong typing encourages and, in most cases, forces the use of nesting. Operations on private types must be defined within the same scope (package) that defines the type. Since the internal structure of that type is not visible outside this scope, all operations must be defined within the scope or the operations must be imported with a generic instantiation.

Circumventing nesting, strong typing, and generics in order to minimize compilation time is a short-term fix with the long-term ramification of decreased reliability and reusability. If the compilation overhead is unacceptable, then alternative Ada development environments will be required.

GENSIM Overview

The GENSIM project was started in 1986 by a group studying ways to increase the reuse of simulator software and the possibility of integrating the dynamics and telemetry simulation capabilities. This group consisted of both software developers and mathematical analysts, all of whom had simulator project experience. At the same time, reuse studies in the Software Engineering Laboratory (SEL) [Solomon, 1987] showed that reusing code without modification (verbatim reuse) yields a tremendous reduction in development cost. One of the early products of the GENSIM effort was a study that estimated that costs of software development for a dynamics simulator could be cut in half by creating verbatim reusable components.

The GENSIM team believed that the best approach to maximizing verbatim reuse was to reuse products from all phases of the software engineering life cycle. This belief was based on developer experience, rather than any formal software engineering theory. The simulator problem was divided into "modules," each of which models an entity in the problem domain. The products associated with each module include a specification, design documentation, code, and test cases; each follows project-wide standards. A module specification consists of a complete definition of the inputs and outputs needed, the algorithms to be implemented to model the entity, and documentation of the mathematical analysis and assumptions underlying the algorithm. A module design is built according to standard protocols for initialization, computations, and the passing of parameters between modules. These protocols allow the individual modules to be configured into a standard simulator architecture.

To determine the feasibility of a generic simulator, a prototype is being developed and applied to a simplified mission. After the prototype demonstrates the ability to configure a dynamics simulator for different missions, a full set of components and modules will be developed.

GENSIM Architecture

This subsection first describes the GENSIM dynamics simulator architecture, then discusses design issues for both modules and standard subsystems. The next subsection discusses lessons learned from the initial GENSIM design that can be applied to the final version of the system.

Figure 9 shows the standard dynamics simulator architecture. The reusable modules are parts of the spacecraft, hardware, and environment models (SHEM) subsystem. Typical SHEM modules include Sun sensor modeling and geomagnetic field modeling. The spacecraft control subsystem is always mission dependent because a dynamics simulator is intended to test attitude control algorithms for a specific satellite. The only reusable parts in this subsystem are a simulated ground command uplink interface and a module that computes estimation and control errors. The user interface has the obvious capabilities of reading and editing input parameters and producing reports and plots from analysis results. The case interface subsystem is responsible for maintaining simulation cases, including analysis results data, simulation input parameters, and suspended simulation cases. The

case interface also standardizes communications between the SHEM modules and the user interface. The simulation executive subsystem has two major purposes: to manage requests to control the simulator and to control the sequencing and timing of module execution. The last subsystem is a utilities subsystem, which consists of several generic units and a set of standard, interrelated instantiations, as is described in Section 2. For example, the instantiation of a generic linear algebra package requires a square root function, which is provided by the instantiation of a general mathematical functions package.

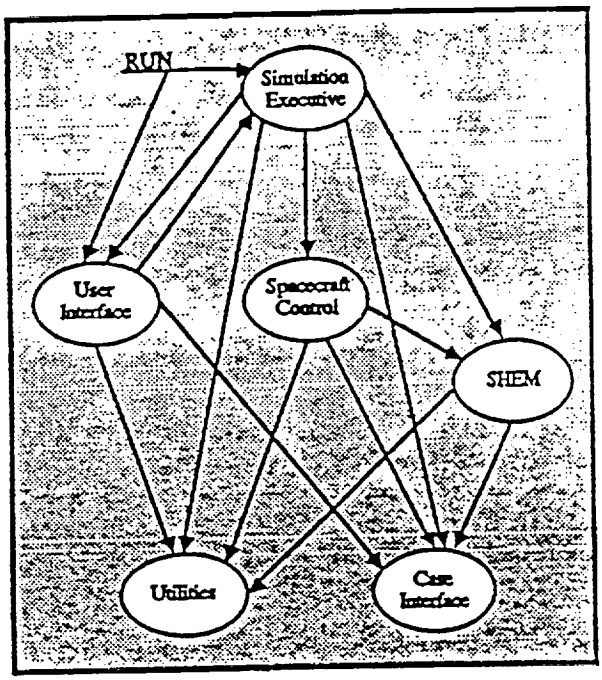

Figure 9.

Figure 10 shows the implementation for a typical SHEM module (Module K). The package Module_K performs the actual modeling. It can be initialized and invoked from the simulation executive, and communicates with other SHEM modules through procedure and function calls. The generic module database and generic module results objects are generic packages that implement the standard communications between a module and the other subsystems. These generics are instantiated with types and values from the Module_K_Types package. These instantiations are called on by the Module_K package when the model itself is being initialized or activated, and they are called on by case interface components whenever parameter or results data is required by some other subsystem. Using this standard approach allows a different set of SHEM modules to be used for each mission.

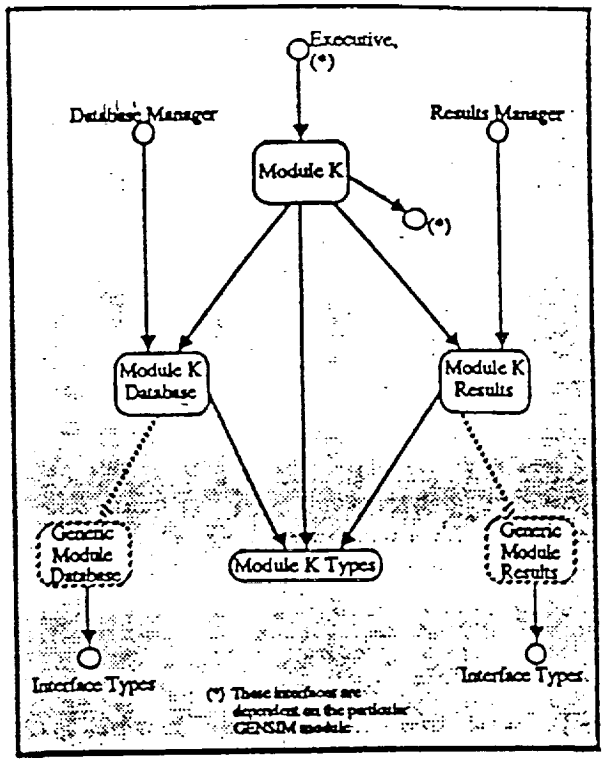

Figure 10.

The GENSIM design just described is a conservative extension of existing simulator designs. In general, the dynamics simulation capability remains the same, but the design has been reworked to be more object oriented. For example, the case interface subsystem was added to treat the concept of simulation case as an object, rather than to distribute those capabilities between the user interface, the SHEM, and the utilities. The utilities subsystem was also changed from one gigantic generic package to several smaller, independent generic units. The main effort in GENSIM has been directed toward generalizing the design to make the components reconfigurable, rather than adding new capabilities.

The user interface, case interface, and the simulation executive subsystems must be implemented as generic subsystems that can be parameterized by the selection of modules for a given mission. The case interface subsystem can be used to demonstrate how a generic subsystem is designed. The discussion of individual modules in the previous paragraph shows how the generic module database and module results packages are used by modules. The case interface subsystem must access these same instances to communicate with the user and to maintain simulation cases.

Figure 11 shows the design for the generic case interface subsystem. The case manager object is responsible for managing simulation cases as a unit; and the ground command interface, parameter interface, and results interface objects manage components (such as analysis results) of a simulation case. The parameter interface and the results interface also manage the communications with SHEM modules. Figure 11 shows that all these packages are interrelated; however, they are used one at a time. For example, a procedure that edits ground commands would use the ground command interface but not the other objects.

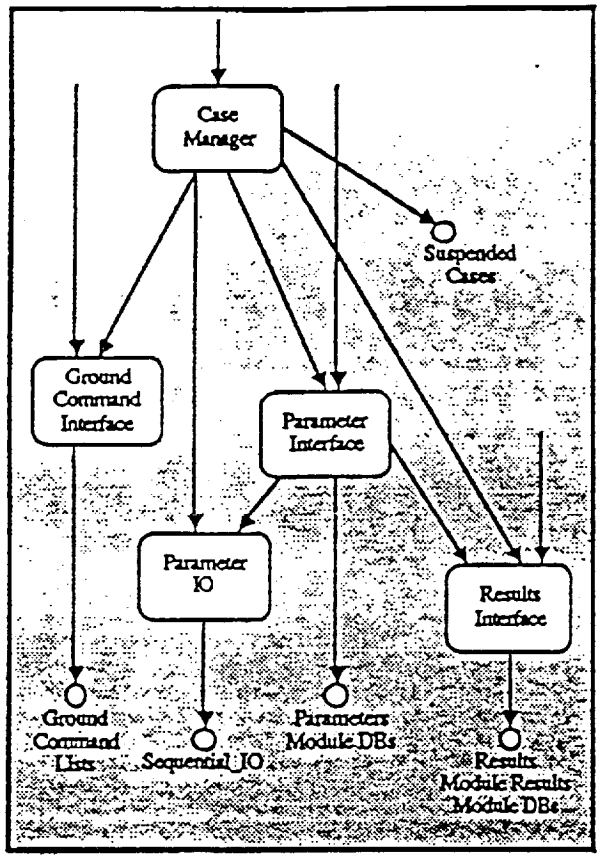

Figure 11.

If these objects were not being implemented as generics, each object in this subsystem would be implemented as a library package, which could be imported independently. With a generic subsystem, each of these packages must be parameterized, and many of the generic formal parameters are common to more than one package. If each object were implemented as a separate generic package, there would be multiple definitions of the same formal parameters with all the maintenance problems such a redundant structure entails. Since the packages in this subsystem are coupled anyway, the case interface subsystem is implemented using the nested generic definitions technique (described in Section 2 of this paper), which allows the common parameters to be placed in the generic part of the composite package.

In the GENSIM design, even the parameters that apply to only one package were placed in the composite package. When this is done, the nested packages are no longer generic. This approach reduces any possible confusion between generic packages and their instances by allowing the user to instantiate the entire subsystem. Then the nested packages can be used without having to instantiate more generics. The cost is that the nested packages are now more highly coupled. In the case interface, this increased coupling is justified because all the coupled packages are part of the abstraction "simulation case." The utilities subsystem consists of independent generic packages, where the coupling is introduced between *instances* of these generics. This approach allows the generic packages to be used outside the context of dynamics simulators. There is no corresponding need to use individual components of case interface outside the context of dynamics simulators because the coupling between the components is defined by the nature of a simulation case. The degree of coupling allowed in the design and implementation of reusable components is one of the key judgments developers must make.

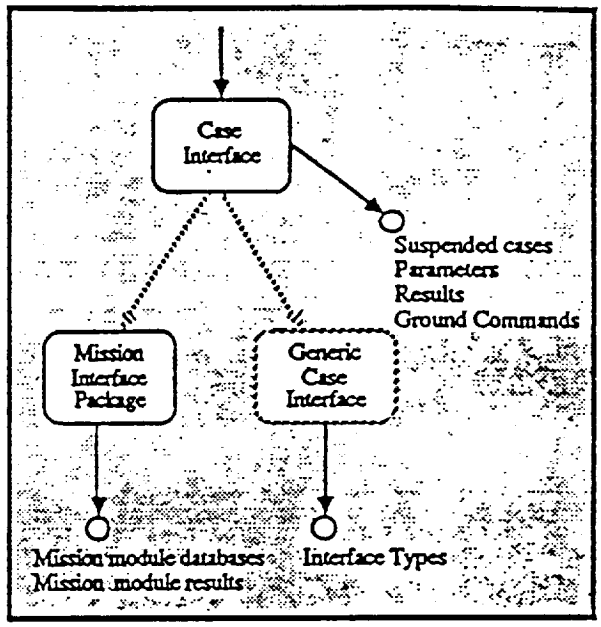

Figure 12.

GENSIM Contributions

The major benefits derived from the GENSIM project are in the categories of (1) gaining experience in the use of advanced Ada features, (2) getting ideas for improvements in simulator design, and (3) producing the reusable components themselves. This subsection will focus on the first two categories. In the first area, the redefinition of the utilities subsystem as a set of *independent* generic packages tied together as a set of *interrelated* instantiations demonstrated the ability to write generic packages as described in Section 2.

When designing a subsystem this way, the developer must make sure that all the generic formal parameters designed are matched by actual parameters provided by some other package. It is also eases development if the code is written and tested bottom up, so that the lower level instantiations needed to instantiate the senior-level generics are tested and, in turn, can be relied on to support the testing of other objects. If care is taken to design a set of generic packages with consistent naming conventions, defaults can be provided by standard instantiations, allowing the rapid writing of instantiations for testing purposes. When all these conditions are met, the techniques described in Section 2 work very well. The decoupled generics/coupled instantiations technique is the best approach to develop packages that provide the ability to use problem domain abstractions rather than predefined Ada constructs.

GENSIM has also contributed to the use of strong typing by using more private types than previous simulators and by beginning to focus on the decision criteria for their use. The criteria for using private types is that they should add the protection of data integrity. For example, the attitude types package defines the private type COSINE_MATRIX. This type is identical in data definition to

any other 3-by-3 matrix but has a set of operations that guarantee that a COSINE_MATRIX always represents a rotation. In the case of GENSIM's orbit data types, a private type does not add any such data protection; the effect is to force a user to use operations provided for the data type in precisely the same way as one would use an assignment statement. When this is the case, the type should be made visible in a package specification.

Possible Improvements to GENSIM

The GENSIM prototyping has been successful in generalizing dynamics simulator designs, but some features inherited from past simulators can be improved on. Currently, simulator module state data types are built from individual scalar objects or arrays of scalar objects. A more object-oriented design would define abstract data types (ADTs) for the problem domain entities and then use these ADTs to define module states. This is particularly true when there are multiple objects of a type, as is the case with spacecraft sensors. As an example, the current design of a fine Sun sensor model defines the simplified module state as follows:

```
package body Fine_Sun_Sensor is
  -- N is the number of Fine Sun Sensors used for a mission
  type STATE is record
    Alpha_Angles
        : Double_Linear_Algebra.VECTOR(1..N);
    Beta_Angles
        : Double_Linear_Algebra.VECTOR(1..N);
    Alpha_Limit
        : Double_Linear_Algebra.VECTOR(1..N);
    Beta_Limit
        : Double_Linear_Algebra.VECTOR(1..N);
  end record;

  Module_State : STATE;
  ...
end Fine_Sun_Sensor_Module;
```

A better implementation is as follows:

```
with Fine_Sun_Sensor_ADT;
package body Fine_Sun_Sensor_Module is
  type STATE is array of (1..N)
    of Fine_Sun_Sensor_ADT.FINE_SUN_SENSOR;

  Module_State : STATE;
  ...
end Fine_Sun_Sensor_Module;
```

In the second implementation, Fine_Sun_Sensor_ADT .FINE_SUN_SENSOR is an abstract data type that encapsulates all the necessary attributes for a single sensor and provides both selector and constructor operations. One advantage of using abstract data types is the tighter encapsulation of data. If a change is made to the fine Sun sensor modeling, the scope of the change is then restricted to the body of the abstract data type package rather than affecting the entire module.

Another advantage of using abstract data types in this context is the strict separation of the problem domain object itself and its use within a software system. In the first example, the declaration of Alpha_Limit mixes the problem domain concept of a limited sensor field of view with the fact that N sensors are used for a particular mission. The second implementation makes it clear that N refers to the number of sensors and that the abstract data type encapsulates each sensor's angles and limits.

When the problem domain object (or class) is implemented with a distinct Ada library unit, it is possible to use the object-oriented programming concept of inheritance to create a hierarchy of classes and subclasses. Figure 13 shows how this could work when all the details of a fine Sun sensor model are considered. This inheritance tree, which is implemented using nested instantiations, shows four levels of increasing complexity, starting with the superclass FSS_ADT and creating a chain of subclasses from there. Each of these four generics can be instantiated either as a library unit or nested within a module. Each subclass in the chain tailors its superclass by incorporating the models provided by the respective utility packages. Inherited operations can also be specialized and new operations can be added to a package [Stark, 1987]. For example, a fine Sun sensor engineering model needs to decalibrate simulated data so that calibration algorithms can be tested. Since this decalibration is specific to fine Sun sensors, the operation would be added to the generic FSS engineering model package, with the noise, biases, and misalignments being provided by the generic measurement utilities.

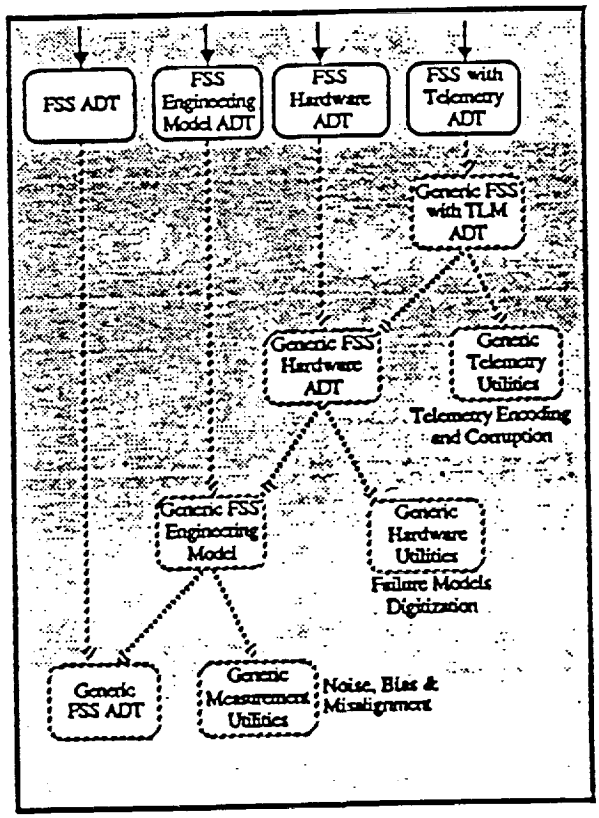

Figure 13.

This approach uses the nested generic instantiations in the same way as UARSTELS. The difference is that all sensor-specific utilities, such as fine Sun sensor decalibration, are part of the sensor abstract state machine, not part of the utility packages.

The use of inheritance allows the selection of an appropriate model for a wider variety of applications. A telemetry simulator would typically pick the telemetry model, a dynamics simulator would pick the hardware model, and error analysis software would use the engineering model. The use of inheritance reduces the redundancy between the different applications, which saves effort in both development and maintenance.

The other area in which the simulator can be made more general is the module types packages. These packages are not defined as generic units, but they contain a mix of mission-specific parameters (such as default initial conditions) and mission independent parameters. The nesting of a generic package within the types package is one possible way to make the system easier to configure. The nested generic would be parameterized by the mission dependencies, with the rest of the types package remaining mission independent. A library instantiation of the nested generic would then be created to define the module's use for a particular mission rather than to extensively modify the types packages. Figure 14 shows how this would work for a fine Sun sensor module. The cost of this is that the other packages in a module now need to import both the types package and the instance of the nested generic, whereas only the types package was needed before. The strict separation of the parameterized part from the consistent part is worth the added complexity.

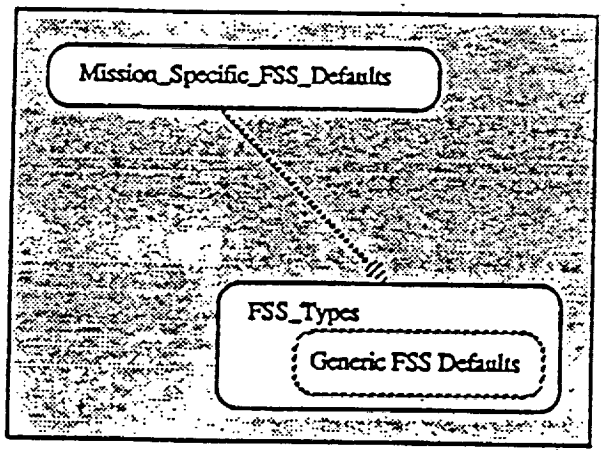

Figure 14.

4. FUTURE DIRECTIONS

The experiences of the UARSTELS and GENSIM projects have demonstrated that the Ada language, and particularly generics, can be used to produce verbatim reusable components that can be fit into more than one architecture. Some other Ada language features need to be examined more closely to see how useful they are for simulation software. There has been a trend to using more strong typing as more experience is gained, but the FDD's Ada software has not gone as far as the Common Ada Missile Packages (CAMP) packages in using distinct types. The CAMP packages use a separate type for each unit of measure, both in generic parameter lists and in nongeneric code [Herr, 1988]. The advantage of using this degree of strong typing is that the compiler is able to catch any dimensionally incorrect computation. The disadvantage is that overloaded operators need to be defined anywhere that two or more different types can be correctly used in a computation. A balance needs to be found between the extremes of CAMP and of using a single floating-point type as the basis of all calculations. To do this, criteria must be defined for the proper use of Ada's typing features. When to use or not to use types, subtypes, derived types, or private types needs clear definition.

This paper's discussion of inheritance focuses on nested generic instantiations as a means of implementing the concept. An alternate approach is to use derived types to simulate inheritance [Perez, 1988]. In the simulators discussed earlier, generics are used for both parameterization and for inheritance. To use derived types for inheritance would require the investigation of the interaction between parameterization and inheritance when different language features are used.

General Concepts for Large-Scale Verbatim Reuse

The lessons learned by the UARSTELS and the GENSIM projects have led us to a general reuse model. This model defines different levels of reuse and which reuse-in-the-small techniques should be applied at which level. Figure 15 shows the leveled reuse model on the left and typical examples on the right. As in most layered models, the higher layers depend on services provided by the lower layers.

	LEVELS	EXAMPLE
ARCHITECTURE LEVELS	• System Templates	Generic_Case Interface
	• Component Templates	Double_Precision _FSS_Module
PROBLEM DOMAIN LEVELS	• Domain Objects and Classes	FSS_ADT
	• Language Extending Objects and Classes	Linear Algebra

Figure 15.

The lowest layer of the model is the *language extension layer*. This layer's purpose is to create a problem-specific language by adding reusable Ada components to the existing capabilities of the Ada language. In the flight dynamics domain, this means defining types and operations for mathematical constructs such as vectors, matrices, and orbits. Applications code can then be developed using the specialized capabilities rather than predefined Ada constructs. This level can be considered the state of the practice for software reuse. The Booch components and the EVB GRACE components are at this level.

The language extension layer itself uses a layered approach. The domain-specific objects are usually built on top of more general objects. The orbit data described above is specific to flight dynamics, but it is represented as two vectors representing position and velocity. When carried into design, an orbit data types package would depend on a more general linear algebra package that exports vector types.

The other important distinction at this level is between entity abstractions and action abstractions. An object with action abstraction is completely described by what it does. A sort package provides operations to sort data; a random number generator generates random numbers. An object with entity abstraction has attributes beyond its set of computations. For example, a queue can be described as a set of homogeneous data that is accessed and modified using a FIFO protocol.

Figure 16 shows how the level of abstraction and level of generality can be used to characterize language extension components. Some typical simulator components are characterized by these two characteristics. The scale from domain specific to general is more continuous than is shown on this diagram. For example, a linear algebra package is specific to the mathematical domain, but it is considered a general-purpose package in the flight dynamics domain. Thus, it would fall somewhere in the middle of the scale. The distinction between entity and action abstraction is more clear cut. If the object has relevant properties beyond the actions it performs, it has entity abstraction. These properties are seen in Ada code as state information that can be retrieved and modified by a package's operations.

	ENTITY ABSTRACTIONS	ACTION ABSTRACTIONS
DOMAIN	Quaternion Orbit	Telemetry Encoding Hardware Failure Models
GENERAL	Stacks, Queues, Vectors, Matrices	Sorts, Integrators, Random Number Generators

Figure 16.

The next level of the model is the *domain level* . This is the level at which the major problem domain entities reside. State-of-the-art reuse libraries such as the CAMP contain components that are reused at this level [Herr, 1988]. Both the domain level and language extension level consist of objects and classes. The difference is that the objects at the domain level *define* the problem domain, and the objects at the language extension level are a means of *expressing* the model for a given problem domain object. The fine Sun sensor abstract data types described in the previous section are all problem domain entities. They are described in terms of vector and matrix algebra, and in terms of standard error sources, telemetry encoding, and sensor failure models. The generic packages for fine Sun sensor data types implement the domain entities *using* capabilities provided at the language extension level.

Figure 13 shows how a mix of domain entities and generic language extensions can be used to build a hierarchy of classes and subclasses. The measurement utilities, hardware utilities, and the telemetry utilities are all language extensions, but they are used in building the problem domain inheritance model.

The next level of reuse is the *component template* level, the level at which generic components are built to fit into a given system architecture. The GENSIM SHEM modules and the UARSTELS sensor models are examples of component templates. Components can be built directly from problem domain objects, or they can provide indirect support. In GENSIM, the SHEM modules will be built around abstract data types, such as those provided for the fine Sun sensors, and the standard module database and module results packages that are instantiated to support the module. In addition to these packages, a standard screen format file is used by the user interface to allow user inputs for each SHEM module. The key distinction is that the component template level defines *all* the components needed to fit a problem domain object into a given system architecture, where the domain level consists of a set of objects that are not constrained by a particular system design, but only by the problem being solved.

The component template level objects are also parameterized, but the emphasis shifts somewhat. The fine Sun sensor abstract data types are parameterized by data types for vectors and matrices and by operations needed to interface with other problem domain objects. The fine Sun sensor module is parameterized by items such as the number of sensors, the default input values, and selections of which inputs a user is allowed to modify. Some values of problem domain parameters may be constrained at this level. Figure 15 gives the example of Double_Precision_FSS_ Module. This module has been constrained to use a particular floating-point data type, but it is still parameterized by the number of sensors and default values.

The top level is the *system template* level. A generic system is a reusable design into which individual components can be fit. Objects at this level are parameterized by the set of components being used in a particular configuration and by any other values that have a system-wide effect. In GENSIM, the parameterization of the generic case interface is related to the particular set of SHEM modules being used. The simulation executive is parameterized both by the set of components being used and by the spacecraft's control modes, which affect how often these components need to be executed.

The two template levels provide the capability of quickly building a software system. Like the language extension and domain levels, the capabilities provided by the lower level are used by the higher one. The key distinction is that the lower two levels give a complete definition of the problem domain, and the upper two levels give a complete definition of a generalized software system architecture. It is important that the problem domain objects be completely independent of particular system architectures. To achieve this, the lower two levels from Figure 15 are grouped as *problem domain* levels and the upper two are grouped as *architecture* levels.

The discussion in this section has focused on design issues, not how Ada should be used to realize these designs. The principles that apply to reuse in the small can be extended to reuse in the large. A developer must still be concerned about a mix of generic packages and their instantiations, and the coupling between components remains a key issue.

In the problem domain level, the only coupling between objects should be defined by the problem. The preferred means of linking objects together is to restrict dependencies to those between library instantiations. One previously mentioned exception to this is the simulation of inheritance. Other relationships can also be simulated through nested generic instantiations or nested generic declarations. An example where nested instantiations are useful is in the case where one object is built from simpler components, as an inertial reference unit (IRU) is built from gyroscopes. The IRU presents a somewhat different interface than a gyroscope, although they are strongly related. The nested generic declarations are useful when alternate models depend on the same objects or types. For example, an orbit types package is parameterized in terms of simple mathematical functions, but they are used by a variety of different models for propagating orbits over time. Rather than nesting instantiations of the orbit types within several different models, the designer can present the models as a set of options that depend on the *same* orbit types.

Importing other library units into generic units is not a problem when used for component templates or system templates. Figure 17 shows where the generic case interface package imports the instantiation of a generic types package. This interface types

package provides standard data types for communication between the user, the stored simulation data, and the SHEM modules. The designer should try to minimize this sort of coupling. In GENSIM, only interface types and a common types package are imported into generics in this manner.

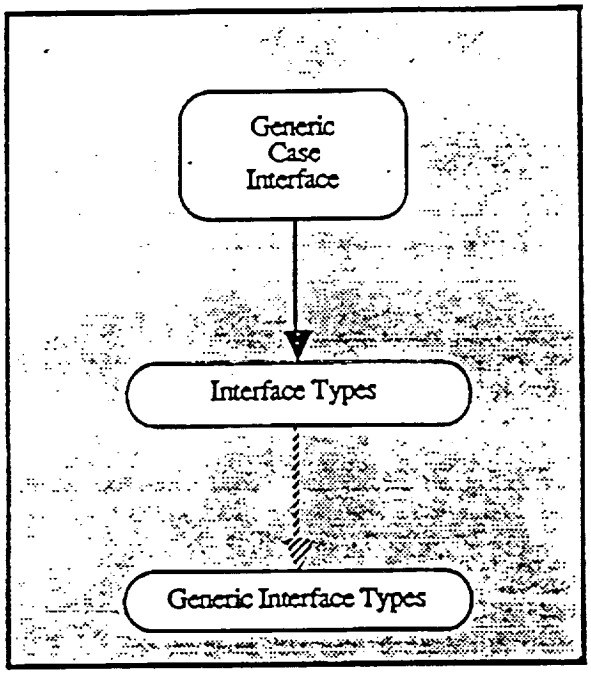

Figure 17.

5. MEASURING THE EFFECT OF LARGE-SCALE VERBATIM SOFTWARE REUSE

This section discusses the impact of the verbatim reuse on project management by describing how costs are affected and the effects of the layered model. A recent SEL study [Solomon, 1987] characterized software components as being new, rebuilt (greater than 25-percent modification), adapted (up to 25-percent modified), and verbatim (unmodified). Expressed as a percentage of the cost of a new component, the costs of the different types of reused components are approximately:

Verbatim	10 %	(actually 7.2 %)
Adapted	30 %	
Rebuilt	50 %	

To make conservative estimates, the 10-percent figure is used for verbatim components, and any nonverbatim component is assumed to be new.

The GENSIM cost study [Mendelsohn, 1988] shows that the current levels of reuse for dynamics simulators save 15 to 20 percent over all-new systems. The study also determined that dynamics simulators have a potential for about 70- to 80-percent verbatim reuse; only the spacecraft control system code is developed from scratch for each mission. These verbatim reuse levels translate to a cost savings of from 60 to 70 percent over an all-new system or at least 50 percent from current systems.

The key to achieving high levels of verbatim reuse is to reuse specifications and design. The analysts who define the requirements for FDD systems developed common mathematical specifications for all systems supporting EUVE and UARS. The current estimate for EUVETELS reuse from UARSTELS is 87 percent, which translates to approximately 80-percent cost savings over a new system. Even the FORTRAN software supporting EUVE has a reuse level of from 60 to 70 percent from UARS, whereas typical levels fall into the 20- to 30-percent range. The increase from reusing the mathematical specifications is much greater than the increase observed as the result of using Ada as the implementation language for simulators [Brechbiel, 1989]. These data confirm the correctness of GENSIM's use of a set of standard mathematical specifications.

In addition to measuring the level of verbatim reuse, the effect of verbatim reuse can be divided into the reuse of problem domain components and the reuse of components at the architecture levels. No FDD simulators have been developed using the proposed reuse model, so the estimate will be based on the fact that the user interface for the dynamics simulators typically contains 40 percent of the source lines of code and no problem domain objects. Since the capabilities of the GENSIM simulation executive and case interface subsystems are currently distributed among other subsystems, 40 percent is a conservative estimate. It is probably correct to assert that the benefits of reusable architectures equal or exceed those of developing reusable problem domain components. It is clear that these benefits are roughly equal.

6. MANAGEMENT RECOMMENDATIONS

The primary management recommendation is to build the problem domain levels first and to build them bottom up. The language extension layer is a means of expression for domain objects and classes. The domain objects and classes serve as the building blocks for reusable system architectures. Another advantage of building the problem domain layers first is the ability to build multiple architectures from the same set of problem domain objects. For simulation applications, this means that the same set of problem domain objects could be used to build a dynamics simulator, a telemetry simulator, or a combined dynamics and telemetry simulator.

The strict separation of problem layers from architecture layers also provides the means of keeping up with technology. The same domain objects would be usable on either an 8086 based computer with a monochrome text screen or on an 80386-based computer with high-resolution graphics. The architecture of the system would be changed, although it would probably not be rebuilt from scratch. The architecture of a system should be driven by technology, and the solution of flight dynamics problems should not be. The separation of these considerations in design makes it easier to manage technological change.

7. CONCLUSIONS

The current state of the art in software reuse is to provide problem domain components and problem domain objects. This paper has demonstrated that designing verbatim reusable components at the architecture level can create approximately the same savings as the current state of the art. The new approach that needs to be applied

to future systems is to strictly separate the problem domain objects from the particular system architectures and to build the problem domain layers from the bottom-up. When this approach is used to develop verbatim reusable software, the resources saved can be applied to new problems (extending the problem domain) or to provide better solutions to existing problems by upgrading the architecture.

REFERENCES

Berard, E., "Reusability Tutorial," *Proceedings of the Washington Ada Symposium*, 1989

Booch, G., *Software Engineering With Ada*. Menlo Park, CA: Benjamin Cummings, 1987

Booch, G., *Software Components With Ada*. Menlo Park, CA: Benjamin Cummings, 1987

Brechbiel, F., "Ada and Specification Reuse Versus Software Cost, Reliability, and Reusability in a Flight Dynamics Support Environment," (to be published)

Ganapathi, M., and G. Mendal, "Issues of Ada Compiler Technology," *IEEE Computer*, February 1989, vol. 22, no. 2, pp. 52-60

Herr, C., D. McNicholl, and S. Cohen, "Compiler Validation and Reusable Ada Parts for Real-Time, Embedded Applications," *ACM SIGAda Ada Letters*, September/October 1988, vol. VIII, no. 5, pp. 75-86

Hudson, W., "Ada Compiler Development," *Defense Science*, March 1988, pp. 59-64

Mendal, G., "Three Reasons To Avoid the Use Clause," *ACM SIGAda Ada Letters*, January/February 1988, vol. III, no. 1, pp. 52-57

Mendelsohn, C., "Impact Study of Generic Simulator Software (GENSIM) on Attitude Dynamics Simulator Development Within the Systems Development Branch," (unpublished study of simulator reuse data)

Perez, E., "Simulating Inheritance With Ada," *ACM SIGAda Ada Letters*, September/October 1988, vol. VIII, no. 5, pp. 37-46

Quimby, K., "Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis," Software Engineering Laboratory, SEL-88-003, December 1988

Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Development Methodology," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

Seidewitz, E., "General Object-Oriented Software Development With Ada: A Life Cycle Approach," *Proceedings of the CASE Technology Conference*, April 1988

Solomon, D., and W. Agresti, "Profile of Software Reuse in the Flight Dynamics Environment," Computer Sciences Corporation, CSC/TM-87/6062, November 1987

Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Life Cycle," *Proceedings of the Joint Ada Conference*, March 1987